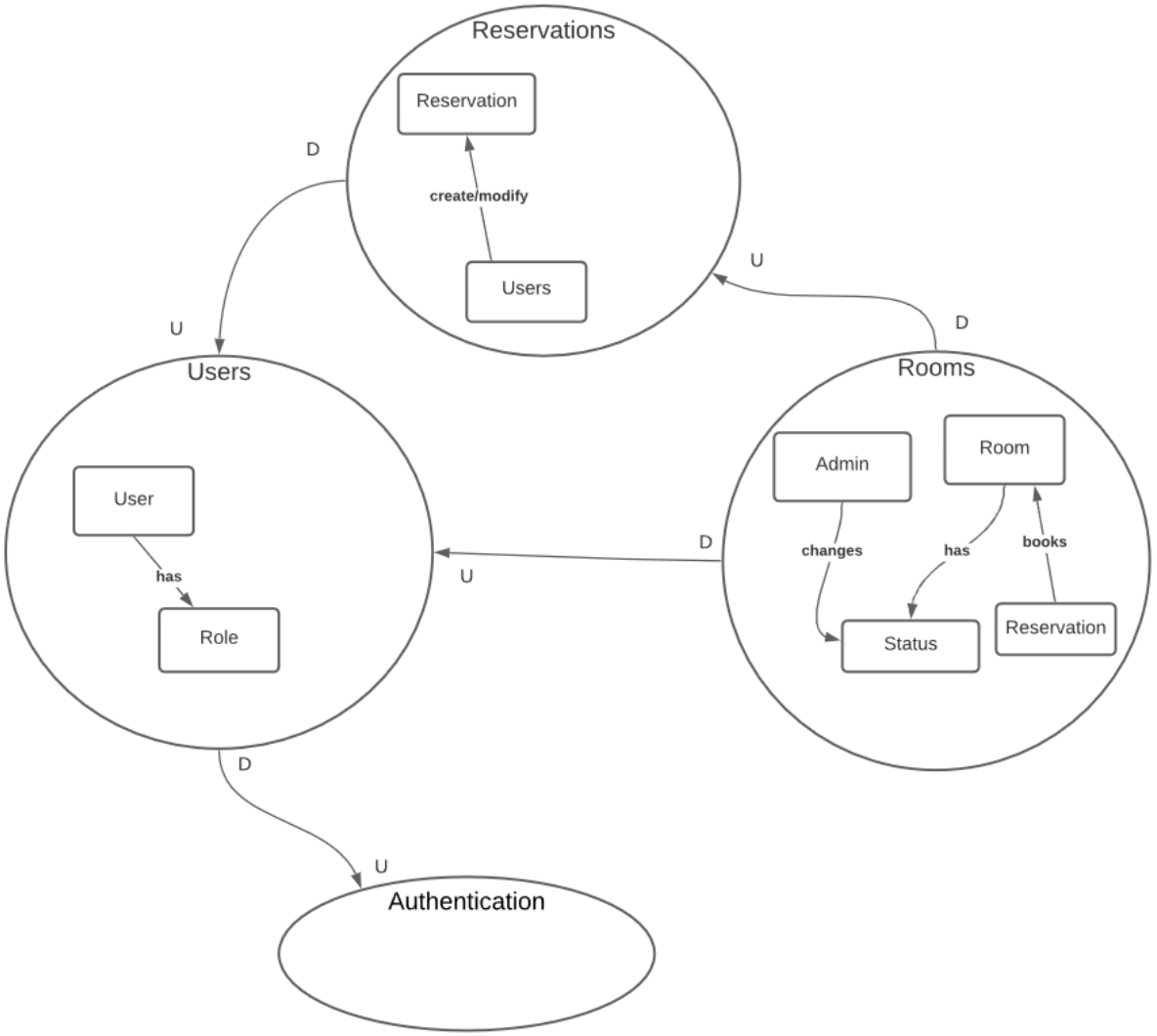
Assignment 1

CSE2215 Software Engineering Methods
2021/2022

Group 11b
Aleksandra Andrasz
Ciprian Stanciu
Matyáš Pokorný
Mike Raave
Milan de Koning
Oanh Tran Chau Kieu

Task 1: Software architecture

The main goal of our application, that we derived from the scenario, is to allow users to reserve university rooms. The application shall feature a simple authentication system to ensure only students and staff are allowed to access the system and only users with certain permissions are able to perform actions like editing reservations or closing rooms.



As you can see in the context diagram above, the bounded contexts we have derived are authentication, reservations, rooms and user. The core domains of our system are rooms, users and reservations since those contexts are the most important part of the system. And authentication is a generic domain, since the functionality is not specific to our system.

We will have 3 microservices in our project: Room, Reservation and User.
We have decided to merge the authentication bounded context into the User microservice because the services would be tightly coupled. As authenticating users requires their data, a User microservice on its own would have little to no separate

functionality. Furthermore authentication mainly provides functionality to the User microservice.

We have chosen these microservices as they cover all the functionality of the program.

We use the Reservation microservice for handling any situation in which a user wants to make a reservation.
Next to that, we use the Room microservice for handling everything concerning a specific room, such as checking opening hours or changing the available equipment.
Lastly, the User microservice handles everything concerning the user of the program, such as defining a role and research groups. The service also handles the authentication of several types of users (employees, secretaries and admins).
The other 2 microservices use this microservice to ensure security.

Microservices

Reservations

The Reservation microservice handles everything related to a reservation. Reservations are stored in the microservice's database, along with identifying information for the user and the room.

Users may reserve a room as long as all the requirements are met, they may change their reservation, while administrators have permissions to edit anyone's reservations.

In order to do this, the service communicates with both the User and the Room microservices in order to validate that the user has the required permissions and that all the requirements are met when creating a reservation. It also communicates with the other services to provide information about existing reservations.

We have chosen to separate the reservations into a separate microservice as it includes a domain and functionality separate from the other services; as one of the most important parts of the entire system, it could not be merged with another microservice.

Rooms

The Room microservice handles all the requests which have to do with a specific room or with multiple rooms. The service's database stores details about the meeting rooms of the university as well as the different buildings where they are located. Aside from this it also stores the equipment located in rooms and information about parts of a room that may require maintenance.

It is connected to the User as well as the Reservation services since the information about a room can be directly requested by a user as well as indirectly via making a reservation.

We have chosen to include the building entity in the Room microservice, as it is never used independently by the system, but only in association with rooms. It makes sense to separate these two into their own service as their domain is distinct from that covered by the other services.

User

The User microservice will store the data of the user and provide functionality for changing it. It will also provide the authentication for the user together with authorization.

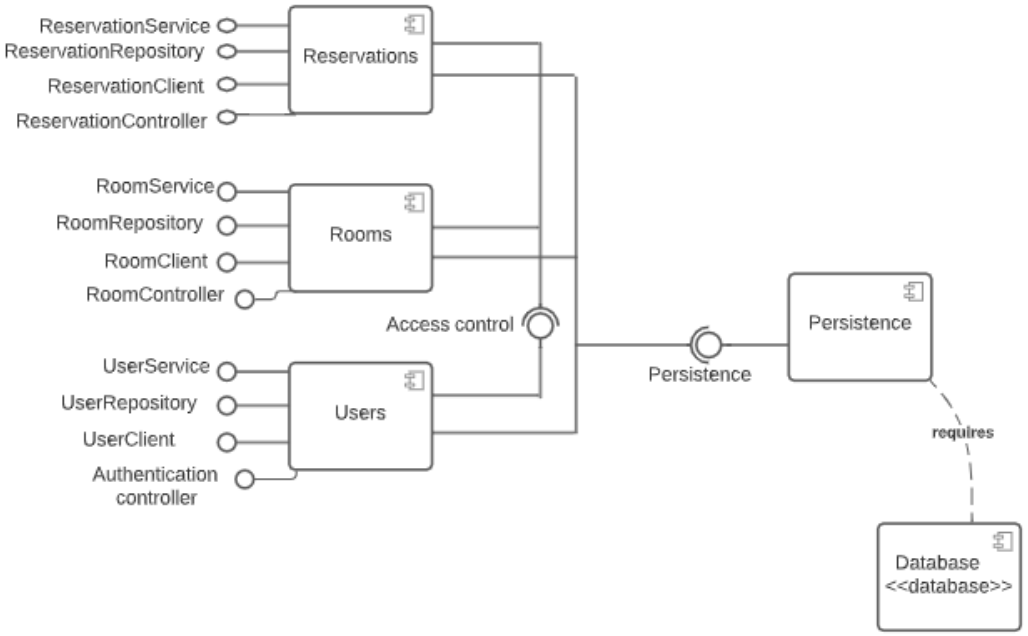
Each user has a unique NetID which they can use to log into the system together with their password. We make sure the password is not stored as plain text for security reasons. Every user also has a role which could be employee, secretary or admin.
Each role has specific actions they are allowed to perform. As an example, while employees can edit their own reservations, admins can edit the reservation of all users. We also store information about research groups with their secretary and members.

For the authentication functionality, when the user opens the system, they should authenticate themselves immediately. After a successful authentication, the user receives a token that they should send with every request. We are going to use JWT tokens to ensure that the system is secure. We have chosen for this method, since a part of the group is already familiar working with JWT tokens and this method should fit the requirements that we want for the authentication system. When other microservices receive the request they will send a request to the user microservice to check the token and return the User object.

The reason we connected authentication functionality within user service, since otherwise the two microservices would be highly coupled. NetId, password and role are kind of shared between bounded contexts, therefore it makes more sense to share them.

Architecture

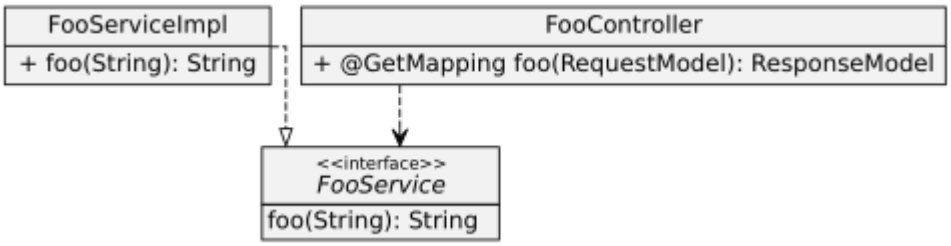
Since we are using microservices, we have a separate database for each microservice. Microservices were designed such that they are as independent of each other as possible. Delegated API calls between microservices are specifically designed to be shallow, and to always request only a singular specific entity. This avoids manual and expensive joining of tables (the so-called “N + 1” problem). Microservices use unique numeric identifiers to refer to specific entities. These IDs are unique across the entire system, because they are only generated by a single responsible microservice.


ReservationService
ReservationRepository
ReservationClient
ReservationController
Reservations
RoomService
RoomRepository
RoomClient
RoomController
Rooms
Access control
UserService
UserRepository
UserClient
Authentication controller
Users
Persistence
Persistence
requires
Database
<<database>>

Task 2: Design patterns

Facade

Our codebase makes a heavy use of the facade design pattern. It is used in multiple places, but the most important use case is in our separation of HTTP handling logic and business logic. Our controller classes themselves do not contain any business logic, and only take care of data parsing and error handling. All business logic is abstracted away in service interfaces, which then have injectable Spring service implementations. This way the controller acts as a facade for the service implementation, exposing the pure Java interface as RestAPI endpoints.



This design pattern allows us to treat services as black boxes, abstracting away the implementation behind a specific interface. Since we can also move our interface into a common library, we can leverage type-checking to ensure our interface implementations are all in sync. This includes the actual business logic implementation in the microservice application itself, as well as the remote API client implementation used for calls between microservices.

To demonstrate this pattern in practice, let us consider the following (simplified) Java code (similar code can be found in our project in all microservices):

```java
interface FooService {
  String foo(String arg) throws NotFoundException;
}

@Service
class FooServiceImpl implements FooService {
  @Override
  public String foo(String arg) {
    return arg;
  }
}

@RestController
class FooController {
  @Autowired
  FooService service;

  @GetMapping("/")
  ResponseModel foo(@RequestBody RequestModel req) {
    try {
      return new ResponseModel(service.foo(req.toString()));
    } catch (NotFoundException ex) {
      throw new StatusResponseException(HttpStatus.NOT_FOUND, "Resource not found!");
    }
  }
}
```

We can see that the service interface allows our controller to ignore all implementation details, and focus on data handling. We can then choose to inject either the true business logic implementation of the service (as the case would be here) or just a service proxy which will perform an external API request, calling another microservice which does implement the business logic.

Chain of responsibility

We implemented the chain of responsibility for filtering a search of rooms. The system requires a set of filters (implemented as a map with for example (“capacity”, 40) as an entry). The system will then check for every filtering criterion whether it is present in the map, and if it is, create another filter in the chain. We choose to use this design pattern so that later on it’s easy to add different filtering criteria.

All of the filter classes extend a BaseFilter class, which has a method to set the next filter in the chain and a handling method that calls the next filter if it is present. The filters are provided with the criterion when being constructed. For example: A capacity filter class will have a parameter that determines the minimum capacity of a room in the constructor, which it can then use to filter out rooms. The head of the chain is always just the default implementation of BaseFilter, so that the chain will still respond normally even if there are no search criteria provided. For this reason, the BaseFilter class was not made abstract. The children of BaseFilter call super.handle(room) to forward the request to the next handler. (Note that at this point in time, the Availability and Equipment filter have not yet been implemented. The implementation of these will happen in these classes themselves and will not modify the pattern in any way.)